DSL course: individual work #1

Tasks

- ✓ create data structure for regular expression syntax tree;
- ✓ function to parse expression into AST;
- ✓ create data structure for non-deterministic automaton(NDFA);
- ✓ create function that transforms regular expression AST into automaton;
- ✓ create an application that receives the representation of regular expression and some word and than checks whether the word satisfies the regexp condition.
- ✓ create data structure for deterministic automaton(DFA);
- ✓ create algorithm for determinization(with removing of unreachable states);
- ✓ create algorithm for minimization.

How to install and run

Clone/download the repository and execute `python lab1_app.py`. There is no dependencies of external libraries, only standard types, type annotations and local packages.

It works on `python3.6`, so please use the same or newer version.

Test set

There are some test sequences inside `test_cases.txt`. Just run `lab1_app.py` and input the filename.

There is `tests.py`, run it, it has other checks and examples. To understand what is going on there, open the script is required.

About the implementation

Syntax of the regular expressions

Regular expressions is a way to define some regular language `L`.

There are rules that are used to define regexps:

- `a` — just any symbol, UTF-8 symbols can be used, `L = {a}`;
- `|` — logical `or` statement (`a` or `b`) (priority **1**): `a|b` — `L = {a, b}`;
- `ab` — is `a+b` there `+` is concatenation (after `a` must be `b`) (priority **2**): `ab` — `L = {ab}`;
- `a*` — `*` is Clini closure (priority **3**), it means that subexpression to which it is performed can occur any number of times(>= 0): `a*` — `L = {e,a,aa,aaa,...}`, where `e` is empty word;
- parenthesis `(` and `)`, as usual, performs prioritization of some subexpression.

So, the following words and so on are allowed:

- `(a)(b)`
- `(ab)*`
- `a|(b*c*)*`
- `ab*|b*a`

Escaped symbols

There is a way to use special symbols `*`, `|`, `(`, `)` as usual symbols in regexp defining, just escape it with `\\` symbol:

- `\*`
- `\(`
- `\)`
- `\|`

As example: regexp `\(ab\)\*` defines language of only one word `(ab)*`.

How to use the `app.py`

The application uses its own format for massive word checking, just create file with your regular expressions and words that should be checked in the following way:

- every line should contain 2 or 3 substrings with colon (`:`) delimiter;
- the first substring is a regular expression. Be careful, for the regular expressions all symbols matter (including spacing);
- second and third substrings are words that should be checked;
- all words for checking separated by a semicolon(`;`);
- second substring contains words that must belong to the regexp language;
- third substring contains words that must not belong to the regexp language;
- you can keep the second substring empty, but colon should be placed.

So, the typical lines of the testing file looks like:

- `(ab)*:;ab;abab:aa;bb` — regexp + true + false words;
- `hii*:hi;hiiii` — regexp + only true words(last colon is not necessary);
- `ab*::b;bb;ba` — regexp + empty + only false words.

It means, that you can't use colon and semicolon in your regexps and words, but it is the limitation of the current verifying application (and the test format), not of the regexp or automatons implementations.

About regular expression parsing

After scanning (it is really simple), parsing is performed. Scanner splits string into list of tokens, and parser recursively separates list of tokens by one of the above operators with respect to their priority and parenthesis, and in this way it extracts AST from the expression.

Note: All the actions of parsing occur on the "first layer" of the expression, and here is what I mean.

Let's say that we have regular expression `(ab)|(c|d)`. The "first layer" of the expression is `group1|group2`, where `group1` is `(ab)`, `group2` is `(c|d)` and `|` is binary operator. Every iteration of the algorithm works only with parts of the first layer (operators inside groups are not visible by algorithm).

About translating

After parsing string to the AST, translation is performed.

There is `translate` function, that performs the following steps if AST is not empty:

- receive children for the node;
- if there are:
  - no children: it must be leaf with only one symbol, create automaton from the symbol using `by_value`

constructor;
    - 1 child: it must be only clini node, receive child automaton (`translate(child)`) and perform `by_clini` constructor on it;
    - 2 children: it can be "OR" as well as "AND", receive children automatons (`translate(child1)` , `translate(child2)`) and perform `by_decision` or `by_concatenation` constructors respectively.

If AST is empty,there is only one automaton: `I={0}, F={0}, T={}` .

About state machines

`automaton` package implements non-deterministic(NDFA) and deterministic(DFA) finite state machines.

At first, let me define all sets that non-deterministic automaton by definition must have:

- `Σ` is a vocabulary of symbols;
- `S` is a set of states;
- `I` is a set of initial(start) states, where `I ⊂ S` ;
- `F` is a set of final states, where `F ⊂ S` ;
- `T` is a set of transitions, where `T ⊂ S x Σ x S` .

Finite is about number of states.

Non-determinism means that from the same state can exist more than one transition with the same "trigger" symbol.

In the other hand, determinism means the reverse.

Transitions of NDFA are stored as dictionary of dictionaries of sets (or in Python notation `Dict[int, Dict[chr, Set[int]]]`), each machine can have any number of start and final states.

Transitions of DFA are stored as `Dict[int, Dict[chr, int]]` , there are no multiple transitions of the same symbol for one state, `0` is always start state, can have multiple final states.

The implementations of machines use iteration mechanism for word verifying, not graph depth search. When some symbol is "putted" into automaton, it tries to transit from current state(s) by existing transition(s) to new state(s). Let me show: there are $C = \{s_1, s_2, ...\}$ — current states of a automaton, `a` — some symbol, `T` — transitions, `nC` — new current states. For every $s_i \in C$ try to get $\{s_i, a, n\} \in T$, and if it exists, `nC = nC U {n}` .

DFA works the same way, but there is only one current state (`C = {s}`) and there are no forks of the same symbol (there are no `{s, a, n}, {s, a, k} ∈ T` , that `k ≠ n`).

About algorithms of NDFA joining

The following rules are applied to join state machines with each other(`OR` is for `|` (decision) and `AND` for `+` (concatenation)):

RegExp	Machine creation
`a ∈ Σ` : `a ∈ RegExp`	$S = \{s_0, s_1\}$, $I = \{s_0\}$ $F = \{s_1\}$, $T = \{<s_0, a, s_1>\}$
$e_1 \in RegExp$, $e_2 \in RegExp$: $(e_1\ OR\ e_2) \in RegExp$	$S = S_1 + S_2$, $I = I_1 + I_2$ $F = F_1 + F_2$, $T = T_1 + T_2$

$e_1 \in RegExp$, $e_2 \in RegExp$: $(e_1\ AND\ e_2) \in RegExp$	$S = S_1 + S_2$, $I = I_1 + I'$ $F = F_2$, $T = T_1 + T_2 + T'$ where: if $I_1 \cap F_1 \neq \varnothing$ than $I' = I_2$ else $I' = \varnothing$ and $T' = \{<s_1, a, s_2> \in S_1 \times \Sigma \times I_2: <s_1, a, s_1'> \in T_1$ for some $s_1' \in F_1\}$
$e \in RegExp$: $e^* \in RegExp$	$S_* = S + \{N\}$, $I_* = \{N\}$ $F_* = \{N\}$, $T = T + T' + T''$ where: $T' = \{<N, a, s_2> \in \{N\} \times \Sigma \times S: <s_1, a, s_2> \in T$ for some $s_1 \in I\}$ $T'' = \{<s_1, a, N> \in S_* \times \Sigma \times \{N\}: <s_1, a, s_2> \in T'$ for some $s_2 \in F\}$

Determinization

The implementation of NDFA determinization uses powerset construction technique(see example chapter). The article above is good, but uses epsilon moves which the implementation haven't, so keep it in mind.

Briefly, there are set of viewed **sets** of states(`V`), queue of state sets(`Q`) and set of new transitions(`nT`). Than the algorithm steps look like that:

1. `V = {}` ;
2. `Q = [I]` , where `I` is initial states;
3. `e = Q.pop()` , if `e in V` than repeat step 3, if `e` doesn't exist go to step 4, else:
   - `V = V U {e}`
   - get all transitions that begin in states of `e` : `T = {start, symb, {ends}}` ;
   - `nT` = `nT U {e, symb, {ends}| where symb and {ends} from T}` ;
   - `Q = Q U {all {ends} from T}` ;
   - repeat step 3.
4. set of initial states is initial in new automaton(can be only one);
5. sets that contain one of the final states are final in new automaton.

Minimization

Minimization is a process of merging of equal states.

Equal states is states that have the same paths to some final states (empty path is also allowed).

The implementation uses fact, that state `s1` equals to `s2` means that if transition for `s1` by symbol `a` exists than transition for `s2` must exist and states to which they transfer must be equal, and vice versa, they are not equal if there is some not equal reachable, or there is no even such transition.

For receiving equality sets, we build lower triangular matrix, where its indexes are states numbers (without dublicates and diagonal(because every state is equal to itself)).

Briefly, it's Hopcroft's alforithm with some improvements.

Algorithm:

1. For every `i` and `j` state, if one of them is final and other is not(XOR), we write 1(not equal, because only final states can have empty path to some final state) in `ij` cell of the matrix.
2. Then, while we had changes in the previous iteration of this(second) step, for every `i` and `j` that have zero in

`ij` cell(possibly equal), we check all states that are reachable from the states by the same vocabulary symbol, and if the reachable states are not equal each other, `i` and `j` is also not equal, so we put 1 (not equal) to `ij` cell.

3. Then in the end we will have matrix, where all equal states are marked by 0. So we can construct equality sets for all the states, and than transform previous transition graph, leaving transitions only between equality sets.